Australian Government
Department of Defence
Defence Science and
Technology Organisation

Developing an Evaluation Method for Middleware-Based Software Architectures of Airborne Mission Systems

Kate Foster[1], Jenny Liu[2] and Adam Iannos[1]

[1] Air Operations Division, Defence Science and Technology Organisation

[2] National Information and Communication Technology Australia (NICTA)

DSTO-TR-2204

ABSTRACT

The Australian Defence Force (ADF) is acquiring airborne mission systems that incorporate component-based and distributed computing systems. Such systems are built on middleware technologies. As DSTO is responsible for technically evaluating ADF acquisitions, one area of research in the Air Operations Division is the evaluation of middleware-based software architectures. In order to conduct this research, DSTO and NICTA have collaborated to extend NICTA's middleware evaluation method and apply it to the airborne mission systems domain.

RELEASE LIMITATION

Approved for public release

Published by

Air Operations Division
DSTO Defence Science and Technology Organisation
506 Lorimer Street
Fishermans Bend, Victoria 3207 Australia

Telephone: (03) 9626 7000
Fax: (03) 9626 7999

AR-014-310
July 2007

APPROVED FOR PUBLIC RELEASE

Developing an Evaluation Method for Middleware-Based Software Architectures of Airborne Mission Systems

Executive Summary

The Australian Defence Force (ADF) is acquiring airborne mission systems (AMS) that incorporate component-based and distributed computing systems in order to enhance its Network Centric Warfare (NCW) capability. These systems are built on middleware, which is a class of software infrastructure technologies that use high-level abstractions to simplify the construction of distributed systems. Middleware architectures play a crucial role in the overall quality of distributed applications.

The Defence Science & Technology Organisation (DSTO) is responsible for evaluating technical proposals for aircraft systems for the ADF. It would be useful to have a method that evaluators of middleware-based systems could use to rigorously assess technologies and determine their fitness for purpose. This would benefit the ADF by uncovering potential design and implementation problems in systems and platforms that incorporate middleware.

The Air Operations Division (AOD) branch of DSTO has, therefore, developed a research program to investigate the evaluation of software architectures of middleware-based systems. In order to perform evaluations as efficiently and effectively as possible, AMS needs to develop a capability in evaluating component-based and distributed software architectures. This includes codification of the evaluation process and reuse of evaluation knowledge from one project to another. Such a capability would promote organisational learning and lead to effective and efficient evaluation of projects.

Researchers from NICTA's Empirical Software Engineering program have developed a structured approach to address the evaluation of middleware architectures, called *MEMS* (Method for Evaluating Middleware architectureS). MEMS is a systematic and rigorous approach for evaluating the various attributes of the architecture of middleware platforms, components and the interfaces for integration with different software applications. NICTA and DSTO have collaborated to extend MEMS to support the evaluation of airborne mission systems that incorporate component-based and distributed technologies. This work was conducted under the AOD Long Range Research (LRR) Task 06/075.

The MEMS extension was applied to the Hybrid Mission System Testbed (MST) at AOD by outlining an evaluation plan. The Hybrid MST provides infrastructure that enables the investigation and demonstration of distributed computing technologies and concepts in modern airborne mission systems. The Hybrid MST is not a simulation of an airborne mission system; rather it incorporates components similar to those found in such systems.

Further research will involve:

- the development of a detailed evaluation plan for the Hybrid MST
- instrumentation and configuration of the Hybrid MST to enable the evaluation plan to be implemented
- conducting experiments for each of the scenarios in the evaluation plan; and
- analysing the results obtained from these experiments.

The evaluation results and the patterns used in the Hybrid MST may be documented using an architecture knowledge management tool also developed by NICTA.

This collaboration has resulted in an improved capability at DSTO to reliably and efficiently evaluate architecture risk during system acquisition. The project also resulted in an enhancement of NICTA's research by providing an industrial environment for technology trial, usage and improvement.

Authors

Kate Foster
Air Operations Division

Dr Kate Foster is a Research Engineer with DSTO's Air Operations Division. Her current work involves support to the Airborne Early Warning & Control acquisition, research into the evaluation of component-based and distributed software architectures, and participation in the DSTO Net Warrior initiative to experimentally investigate aspects of net centricity. Kate obtained a Bachelor of Engineering (Electrical and Electronics) (Hons) and a PhD from Swinburne University in Melbourne, Australia.

Jenny Liu
National Information and Communication Technology Australia (NICTA)

Dr. Yan Liu obtained her PhD degree from University of Sydney in 2004. She is a senior researcher in NICTA ATP laboratory, working under Managing Complexity Theme. Her current research involves software architecture evaluation and adaptive middleware. She published her research on software engineering and middleware systems on international journals and conferences including IEEE TSE, JSS, ICSE, OOPSLA, WICSA and CBSE. Her research interests include software architecture, component- and middleware based systems, software performance prediction, and autonomic computing.

Adam Iannos
Air Operations Division

Adam Iannos obtained a Bachelor of Engineering (Computer Systems) (Hons) from the University of Adelaide, Australia in 2001. In 2002, he joined the Air Operations Division at DSTO as a Professional Officer and his work involves support to the Airborne Early Warning & Control acquisition, New Air Combat Capability acquisition, distributed computing design and analysis, and investigating aspects of net centricity.

Contents

Acknowledgements

Some of the information presented in this report was sourced from in-house documentation prepared by Brad Tobin, Peter Temple, Trent O'Connor and Robert O'Dowd of DSTO and discussions with Derek Dominish of Boeing Australia.

Abbreviations

ACE	ADAPTIVE Communication Environment
ADAPTIVE	A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment
ADF	Australian Defence Force
ALMA	Architecture Level Modifiability Analysis
AMS	Airborne Mission Systems
AOD	Air Operations Division
API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
CCM	CORBA Component Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial-Off-The-Shelf
DOC	Distributed Object Computing
DSTO	Defence Science & Technology Organisation
GUI	Graphical User Interface
IDL	Interface Description Language
LRR	Long Range Research
MEMS	Method for Evaluating Middleware architectureS
MST	Mission System Testbed
NCW	Network Centric Warfare
NICTA	National Information and Communication Technology Australia
OCC	Optimistic Concurrency Control
OMG	Object Management Group
OO	Object Oriented
ORB	Object Request Broker
RFP	Request for Proposals
SAAM	Software Architecture Analysis Method
SAF	Software Architecture Framework
SEI	Software Engineering Institute
SRL	System Readiness Level
STAGE	Stimulation Toolkit and Generation Environment
STL	Standard Template Library
TAO	The ACE ORB
TDF	Tactical Display Framework
TRL	Technical Readiness Level
WSM	Weighted Scoring Method

1. Introduction

Modern weapon systems continue to increase in complexity for two main reasons. The first is due to the focus on enabling joint and multinational operations by enhancing the Network Centric Warfare (NCW) capability [DFW 2004] of the Australian Defence Force (ADF). The second is due to the necessity, for the corporate sector, to continue to innovate in the field of weapons [Paravisini 2003]. Furthermore, the growth in the complexity of software has far outstripped the increases in the complexity of hardware components and their connectivity. For example, in the 1960s less than 10 percent of the functionality of the F-4 fighter was based on software; in the F/A-22 it is more than 80 percent [Wait 2006].

To manage software complexity, the Component-Based Software Development paradigm has been developed, under which software systems are constructed from existing or newly created components that may be distributed over a network. The integration of these systems has also become more challenging, particularly for real-time and embedded distributed systems. Many factors (e.g. network latency, predictability, concurrency, scalability and partial failures) need to be taken into account when designing such systems [Voelter *et al.* 2005].

One of the critical factors for the success of system integration is an appropriate architectural design of the system [Clements *et al.* 2001]. This aims to ensure that the system satisfies its key behavioural requirements and quality attributes, such as real-time performance, reliability, security and maintainability.

Since the mid 1990s, developers of complex systems have begun to recognise the need for improved architecture modelling and analysis approaches that enable the process of building systems to be more predictable. Integrating complex systems can be difficult due to emerging properties (e.g. scheduling, fault tolerance and security). System lifecycles are becoming evolutionary as components of systems are upgraded to avoid obsolescence. However, the impact on the system as a whole needs to be considered before upgrades are commenced [Allen *et al.* 2002].

In parallel with the increasing complexity of weapon systems, software intensive acquisition projects have come to be considered the most risk prone in the Defence domain. Such projects often incur schedule delays, cost overruns and reduced functionality [DMO 2004]. The need to address system integration in Defence has become compelling. Research programs have been established to investigate techniques and frameworks for identifying risks. Technical Readiness Levels (TRLs) and System Readiness Levels (SRLs) address some issues [Smith *et al.* 2004], but are not sufficient for identifying system integration risk as they do not provide an explicit risk framework [Nandagopal 2006].

A software architecture evaluation framework can provide a process for identifying the technical risks of a proposed architecture. Every software system has an architecture that serves as a foundation for how the system is developed and its elements are integrated to deliver required functionality and qualities [Bass *et al.* 2003]. The earlier risks are identified, the greater the probability that the system will be cost effective, delivered to schedule and perform as required.

The work in this report was conducted under the Airborne Mission Systems (AMS) Branch Long Range Research (LRR) task (06/075) in collaboration with National Information and Communication Technology Australia (NICTA). This report documents the development of an evaluation method for middleware-based software architectures of airborne mission systems. This includes codification of the evaluation process and reuse of evaluation knowledge from one project to another. Such a capability promotes organisational learning and enables effective and efficient evaluation of projects.

Using NICTA's middleware-based architecture evaluation research, the project focused on the application of software architecture evaluation techniques and tools to the domain of airborne mission systems. An evaluation method for a generic middleware-based airborne mission system was developed and applied to the Hybrid Mission System Testbed (MST) at the AMS Branch of DSTO.

This report is organised as follows:

- Section 2 discusses middleware architectures and their impact on system quality.
- Section 3 reviews related work from three aspects: software architecture evaluation methods, middleware evaluation approaches and the use of scenarios for architecture evaluation.
- Section 4 presents the generic MEMS approach, before it was modified for the airborne mission systems domain.
- Section 5 presents the extension to MEMS that was developed in order to evaluate middleware-based airborne mission systems.
- Section 6 overviews the architecture and components of the Hybrid MST and discusses the particular configuration of components to be used for the evaluation.
- Section 7 outlines an evaluation plan to apply MEMS to evaluate the Hybrid MST, which establishes the base for the next phase of the evaluation.
- The report is summarised and the next phase of the evaluation of the Hybrid MST is discussed in section 8.

2. Middleware and System Quality

Middleware architectures play a crucial role in determining the overall quality of many distributed applications. Middleware refers to a broad class of software infrastructure technologies that use high-level abstractions to simplify construction of distributed systems. This infrastructure provides a distributed environment for deploying application-level components. These application components rely on middleware to manage their lifecycle and execution, and to provide off-the-shelf services such as transactions and security.

Consequently, the application component behaviour and middleware architecture are tightly coupled, and middleware plays a critical role in achieving the quality attribute requirements of distributed applications. If the middleware architecture is poorly designed or implemented, contains subtle errors, is inefficient or lacking in features, it may eventually lead to the failure of applications in meeting their intended requirements.

An evaluation method for middleware-based applications would, therefore, be useful to rigorously assess a technology and determine its fitness for purpose for an application. Such a method would also benefit the ADF, which could use this approach to uncover potential design and implementation problems in systems and platforms that incorporate middleware.

Middleware creates new challenges and issues for software architecture evaluation methods. Firstly, middleware technologies are *horizontal* in nature, providing mechanisms for a wide range of applications in many vertical application domains. The business goals for an application from the perspective of stakeholders are more likely to address the domain-specific application behaviour rather than the requirements for the middleware itself. This indicates that evaluation methods for middleware should be driven by the concerns of individual quality attributes within the scope of specific business goals.

Secondly, the ability of a middleware technology to support given quality attributes depends on the mechanisms and services provided by the infrastructure. Middleware technologies normally provide the flexibility of different mechanisms to address the same quality attribute. For example, different design patterns can be supported to provide concurrency and each has implications for a number of attributes, such as liveliness, performance and scalability. This indicates that evaluation methods require detailed technical input regarding the middleware infrastructure, including its programming model, application programming interfaces (APIs), configuration and deployment. This kind of knowledge helps to identify the effect of different middleware architectures on quality attributes.

Thirdly, middleware technologies are becoming increasingly complex. They typically have several thousand API calls and a collection of integrated services and tools of varying importance to different applications. This makes it difficult to evaluate the quality of a complete middleware technology. Evaluation methods need to be flexible and able to quickly provide feedback on alternative middleware architectures with respect to multiple quality attributes.

3. Software Architecture Evaluation

3.1 Scenario-Based Architecture Evaluation Methods

Software architecture evaluation methods and techniques have been widely studied. These methods and techniques have focused on understanding the relationship between software architecture and one or more quality attributes to ensure that the system ultimately achieves its quality goals while still supporting its functional requirements. A review of these techniques can be found in [Ali-Barbar & Gorton 2004] and Chapter 6 of [Bass *et al.* 2003].

Scenarios are defined to understand how a software architecture responds with respect to attributes such as maintainability, reliability, usability, performance and flexibility. Examples of scenario-based methods are the Software Architecture Analysis Method (SAAM) [Kazman *et al.* 1994], Architecture Tradeoff Analysis Method (ATAM) [Kazman *et al.* 1998] and Architecture Level Modifiability Analysis (ALMA) [Bengtsson *et al.* 2004]. The remainder of this section considers these methods in terms of their inputs, outputs and the roles involved.

SAAM deals mainly with maintainability. Its inputs are software architecture descriptions and quality requirements from stakeholders. SAAM normally investigates and collects requirements from stakeholders using interviews.

ATAM is a two-phase method. The inputs to the first phase include general scenarios (or requirements from stakeholders), software architecture design documentation and the formation of the evaluation team. The tasks in the first phase are to transform general scenarios into specific scenarios and evaluate the software architecture against specific scenarios. The second stage of ATAM presents the results to stakeholders, who provide the business goals, and matches the software architecture with the business goals to analyse the impact of architecture changes based on each scenario. ATAM also deals with multiple quality attributes. ATAM collects the requirements of stakeholders in brainstorming sessions on the scenarios related to the business goals.

PASA [Williams & Smith 2002] is similar to the first stage of ATAM and is dedicated to the evaluation of performance of software architectures. PASA applies software performance engineering to the analysis step of the first phase of ATAM. PASA focuses on performance evaluation and its inputs include use cases, the software architecture to be assessed and the performance related scenarios derived from use cases.

Mature approaches such as ATAM and SAAM have both technical and social aspects. The technical aspects deal with the collection of data and analysis techniques, while the social aspects involve interaction among stakeholders, software architects and evaluators. Technically, MEMS targets middleware, which is a component of the overall software architecture to be evaluated. Therefore it demands more inputs to support the techniques, tools and mechanisms to evaluate middleware architectures and technologies. Consequently, the roles involved in MEMS are more technical and require architects and designers who have considerable knowledge and experience with the use of middleware [Gorton *et al.* 2003].

The relationship between architecture evaluation methods and software development processes is explored in [Nord & Tomayko 2006]. Architecture-centric approaches, such as ATAM, can be applied to the analysis and testing phases of extreme programming activities. Design analysis using ATAM provides early feedback for understanding architectural tradeoffs, decisions and risks. MEMS enhances the development process by emphasising quality attributes and focusing on architectural design decisions in projects where middleware is evolving rapidly to support emerging technologies and agile development methods are applied.

3.2 Evaluation of Middleware

The i-Mate process [Liu & Gorton 2003] has been applied to evaluate Commercial-Off-The-Shelf (COTS) middleware technologies, particularly for the acquisition of middleware for enterprise applications [Nord & Tomayko 2006]. i-Mate is similar to the first phase of ATAM, and requires stakeholders to input business requirements for the middleware to be acquired. The evaluation of performance and scalability is conducted in a laboratory environment by running a predefined benchmark application on all candidate middleware.

Both i-Mate and MEMS require techniques specific to middleware infrastructure, as prototyping with the middleware is essential to conducting the assessment. MEMS is different from i-Mate in that MEMS is concerned with evaluating alternative solutions using a single middleware infrastructure. Business goals are imposed on the output of MEMS and are not a portion of the method. The evaluation is driven by concerns about the quality attributes for specific designs using middleware. In this sense, MEMS is more lightweight and agile than i-Mate.

Methods and techniques are also available to evaluate specific quality attributes of middleware systems [Gorton *et al.* 2003; Kanoun *et al.* 1997]. Quantitative quality attributes, such as performance and availability can be assessed through measurement, analytical modelling and simulation. For example, Tang *et al.* [2004] presented an availability model and analysis method for Sun's Java Application Server, Enterprise Edition 7. The study applied Markov reward modelling techniques on the target software system and estimated the model parameters from laboratory or field measurements.

3.3 Use of Scenarios for Architecture Evaluation

Scenarios have been used in several disciplines, e.g. military and business strategy, and decision making. The software engineering community initially used scenarios in user-interface engineering, requirements elicitation and performance modelling. More recently, scenarios have been used in software architecture evaluation [Bass *et al.* 2001]. Scenarios are effective for software architecture evaluation because they are flexible and, therefore, can be used to evaluate most quality attributes. For example, scenarios that represent failure can be used to examine availability and reliability, scenarios that represent change requests can be used to analyse modifiability, scenarios that represent threats can be used to analyse security and scenarios that represent ease of use can be used to analyse usability. Also, scenarios are

usually concrete, which enable the user to more easily understand their effect [Boehm & In 1996].

The software architecture community has developed different frameworks for eliciting, structuring and classifying scenarios. These include a two dimensional framework to elicit change scenarios [Lassing *et al.* 1999], a generic three dimensional matrix to elicit and document scenarios [Kazman *et al.* 2000] and a six elements framework to structure scenarios [Bass *et al.* 2003].

The scenario generation framework in Table 3-1 is used in MEMS to generate scenarios during scenario development activities. This framework provides a systematic way of capturing and documenting general scenarios, which can be used to develop concrete scenarios and to select an appropriate reasoning framework to evaluate the software architecture.

Table3-1. Six elements scenario generation framework. Adapted from [Bachmann et al. 2003]

Elements	*Brief Description*
Stimulus	A condition that needs to be considered when it arrives at a system
Response	The activity undertaken after the arrival of the stimulus
Source of stimulus	An entity (human, system or any actuator) that generates the stimulus
Environment	A system's condition when a stimulus occurs, e.g. overloaded or running
Stimulated artefact	Some artefact that is stimulated; it may be the whole system or part of it
Response measure	The response to the stimulus should be measurable so that the requirement can be tested

It is important to note that the term *scenarios* in software architecture is different to that used in object oriented (OO) design methods, in which it generally refers to use-case scenarios (i.e. scenarios describing system behaviour). Instead, quality sensitive scenarios describe an action, or sequence of actions, that might occur with the system to be built by using a particular architecture. For example, a change scenario might describe a certain maintenance task or a change to be implemented [Liu & Gorton 2003].

Scenarios used in software architecture evaluation are classified into various categories, e.g. direct scenarios, indirect scenarios, complex scenarios, use case scenarios, growth scenarios and exploratory scenarios [Dobrica & Niemela; Gorton *et al.* 2003; Lawlor & Vu 2003]. The Software Engineering Institute (SEI) has collected general quality attribute scenarios that are intended to encompass all of the generally accepted meanings for quality attributes [Barbacci *et al.* 1995]. A general scenario is, in effect, a template for generating a specific quality-attribute scenario. For example, two (abbreviated) modifiability general scenarios are: (1) changes to the platform occur and (2) additional distributed users arrive at the system.

Since not all of the general scenarios for a particular quality attribute will be relevant to a particular system or class of systems, the analyst must identify those that should be considered and make them system specific [Kanoun *et al.* 1997]. General scenarios can also be categorised according to domain specific software change categories to help the analyst identify those general scenarios that are relevant and need to be made system specific with the help of stakeholders for a particular type of application.

4. MEMS

MEMS is a scenario-based method for evaluating multiple quality attributes of middleware architectures. Similar to other scenario-based evaluation approaches such as SAAM and ATAM, MEMS is founded on key scenarios that describe the behaviour of a middleware architecture with respect to particular quality attributes and in particular contexts. The quality goals and their expression in the form of key scenarios drive the evaluation process. MEMS defines the evaluation process in seven steps, which are described below. MEMS outputs the ratings of each architecture against the quality attributes of interest. The seven steps of MEMS along with the artefacts produced at each step of the middleware evaluation are depicted in Figure 4-1.

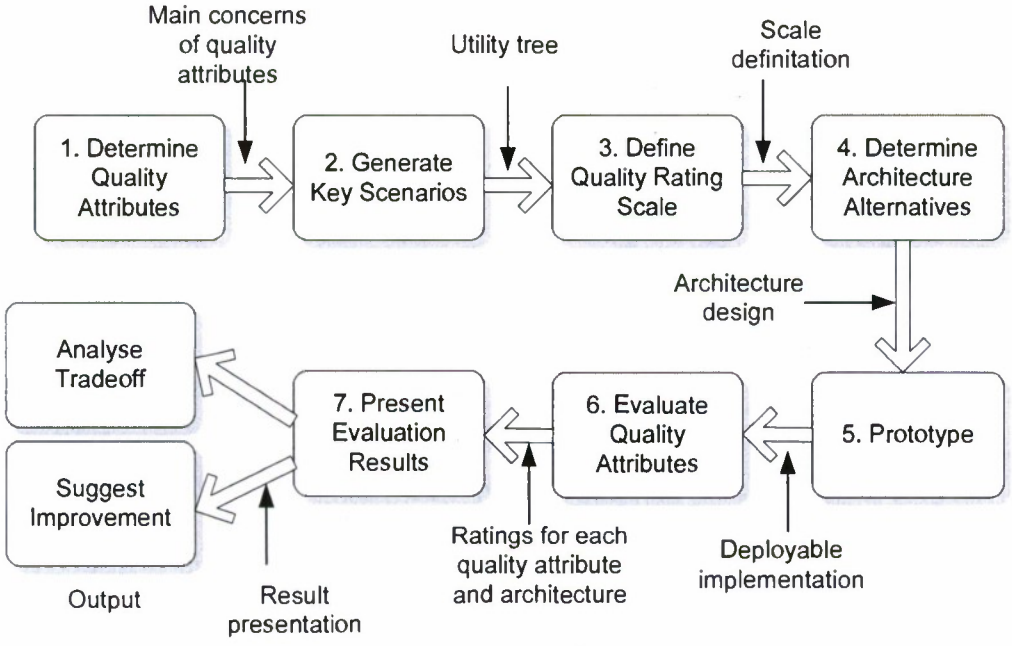

Figure 4-1. The steps involved in MEMS

Step 1. Determine Quality Attributes

The first step is for the evaluator to determine the quality attributes of interest. As discussed in Section 3, one aspect that differentiates MEMS from SAAM, ATAM, and i-Mate is that MEMS is not driven by quality requirements derived from the business goals. Instead, it addresses general quality attributes, such as performance, availability, scalability and security. The main concerns for a quality attribute must be specified within this step. One quality attribute may embody many specific concerns. For example, secure communication can be considered from four different views: privacy, integrity, authentication and authorisation. The purpose of defining the general quality attribute concerns is to set the context for the next step which generates key scenarios for each quality attribute.

Step 2. Generate Key Scenarios

Similar to other scenario-based evaluation methods, scenarios are adopted as the descriptive means to capture concrete quality attribute requirements, as quality attributes by themselves are too abstract for analysis. Real scenarios are developed for each identified attribute or its associated sub-concern. This step also involves organising scenarios and quality attributes. A practical approach for this is the ATAM utility tree (see Chapter 11 in [Bass *et al.* 2003]), which uses quality attribute names as an organising vehicle.

Step 3. Define Quality Attribute Scale

Quantitative attributes can be evaluated using measurement, analytical modelling and simulation techniques. For qualitative attributes, one common approach to consolidating evaluation results is the Weighted Scoring Method (WSM) [Kontio 1996]. WSM requires a clear and unambiguous definition of a rating scale, so that evaluators can give weights or scores to qualitative attributes with respect to the middleware architecture. This step is important so that evaluators have consistent rating criteria.

Step 4. Determine Architecture Alternatives

This step lists the alternative middleware architectures possible for the implementation being considered. Middleware provides multiple mechanisms and services to support the same functionality. Different mechanisms and services can be combined with patterns and frameworks to form middleware architectures. Hence one scenario usually has several alternative architecture solutions.

Step 5. Prototype

A prototype implementation is produced, one for each of the alternative architectures. This step requires skill in programming using the middleware infrastructure as well as knowledge of the techniques used in the middleware. A prototype is executed and measurements are taken for quantitative attributes of interest. Prototyping is also useful to obtain feedback on the architecture design and understand how it may impact other qualitative attributes.

Step 6. Evaluate Quality Attributes

Evaluation techniques from the literature are applied for evaluating individual quality attributes. For quantitative attributes, the evaluation focuses on producing the metric values for a quality attribute, such as transaction response time for performance. Various techniques are available based on analytical modelling, simulation or prototype measurement. For qualitative attributes, evaluators give ratings for each architecture against the rating scales defined for each quality attribute.

Step 7. Present Evaluation Results

The output of MEMS represents the ratings of each potential solution architecture against the quality attributes of interest. The results can be utilised from several different perspectives. They can be further input to other scenario-based architecture evaluation methods, such as ATAM for the trade-off analysis of quality attributes, used to provide feedback to developers of the middleware infrastructure to further improve the middleware, or used to evaluate whether the middleware architecture can fulfil the quality requirements of the application to be built. The evaluation results are visually presented in a way that clearly identifies the ratings of each middleware architecture with regard to individual quality attributes.

MEMS is a lightweight approach as it only concerns interactions between two roles, namely software architect and developer. The role of software architect deals with the activities from steps 1 to 4. The developer provides the expertise and the programming skills for developing the prototype in step 5. The developer has the experience to know the mechanisms and services from the middleware infrastructure that can support the quality goals defined for each architecture alternative. The software architect and the developer then work together on step 6 to evaluate quality attributes. The developer provides feedback on the definition of the rating categories and helps ensure it is clear and unambiguous. The software architect may return to step 2 to refine the category definitions based on comments from the developer.

The architect then produces an evaluation form with the quality rating left blank for each architecture. The form includes the required quality goals, scenario and architecture descriptions, and the definition of the quality rating category. The developer will fill in the evaluation form with ratings for each quality attribute and architecture against the criteria defined in step 3.

The architect can further present the evaluation form to others who have equivalent knowledge, skills and experience as the developer in the evaluation team, and get them to fill in the form. Hence, the role of developer may be filled by more than one person. With different developers assessing the architecture alternatives, a wider range of opinions can be canvassed. If the opinions are inconsistent, the architect needs to further check the rating category definitions that the architecture alternatives are described clearly, and the developers have a clear understanding of the middleware infrastructure being used.

5. Extending MEMS

5.1 Integrating Design Patterns

Developers of middleware-based systems can now employ abstractions, such as design patterns and architecture styles, as guidance when designing and implementing software architectures. These abstractions provide generic guidance regarding the implications of each design pattern on one or a set of quality attributes. For example, the pattern definition templates used in [Schmidt *et al.* 2000] have a *consequence* section to discuss the consequences of each design pattern.

The solution proposed by a design pattern is generic to a type of problem and it has platform independent descriptions of the pattern structure. However, the implementation of this pattern is platform specific and encompasses variants to the generic descriptions. Many decisions that impact quality attributes are embedded in a particular pattern. Software architecture design with patterns needs to adapt the pattern to enable its use in a particular context.

MEMS provides a flexible architecture evaluation method for middleware-based systems that can integrate patterns with other artefacts, such as quality attributes and scenarios. This section describes the extension to MEMS for evaluating the impact of alternative implementations of design patterns on quality attributes.

Figure 5-1 illustrates the relationship between patterns and other artefacts of MEMS. The use of patterns is an architectural design decision and a pattern is implemented to satisfy the scenario defined for a particular quality attribute. A scenario has a set of responses and stimulus and inherently these responses and stimulus are applied to patterns. Responses and stimulus can be modelled by metrics of the quality attributes. A pattern also has a context, problem domain and rationale behind the solution. This means that when the stimulus is constructed as an input to the system under evaluation, the evaluation should be devised within the context of the design pattern.

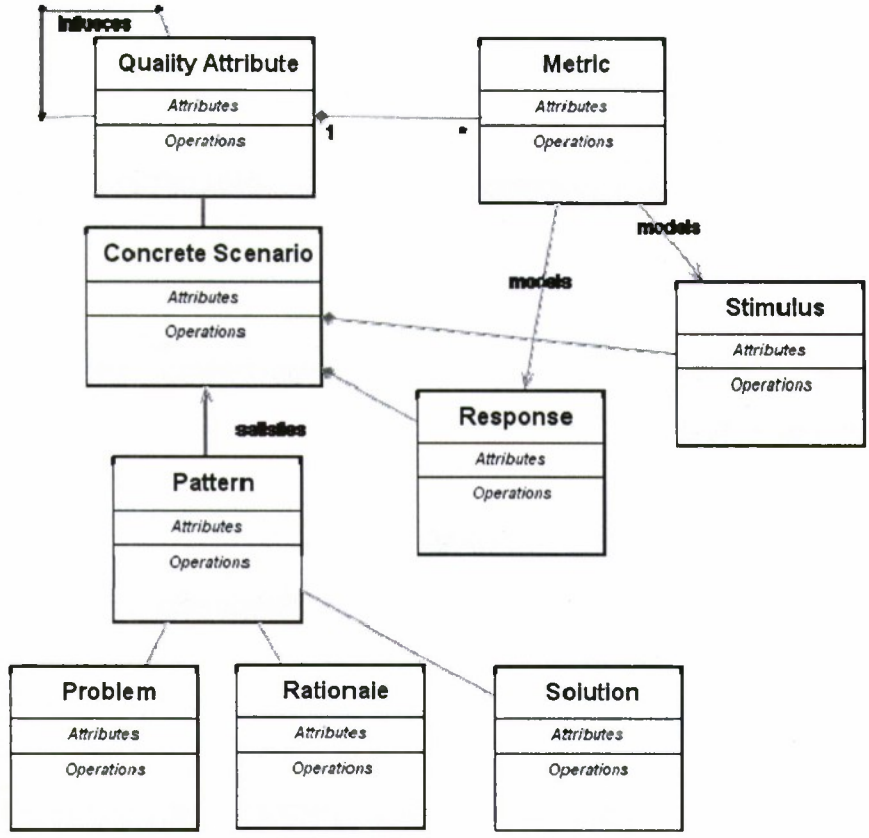

Figure 5–1. Patterns, quality attributes and metrics

The generic MEMS approach is extended to integrate design patterns as shown in Figure 5–2. As discussed above, architectural design decisions are embedded in design patterns. Step 4 in MEMS *Determine architecture alternatives* (Section 4) is now customised as *Determine pattern alternatives*. This step is further elaborated by three sub-steps:

- List quality attributes affected by the design pattern. This is achieved by examining the design pattern descriptions for the structure (elements involved), dynamic (how elements interact with each other) and consequences in a general context. The affected quality attributes are described by the key scenarios produced by step 2 of MEMS. The key scenario needs to match the problem and solution specification of the design pattern.
- Determine the metrics for measurement. The quality attributes need to be described using metrics so they can be evaluated through the measurement of these metrics. The metrics can be either qualitative or quantitative. For example, the performance quality attribute can be described by a quantitative metric such as response time, while the programmability (or modifiability) quality attribute is more often evaluated by qualitative measurement.

- Identify pattern variants. A scenario can be supported by more than one pattern and an individual pattern can also have a number of variants, such as applying different communication protocols between two elements. The pattern structure, dynamic and implementation descriptions can help to identify alterative design decisions using patterns. For example, the *active object* pattern [Schmidt *et al.* 2000] can be applied to improve concurrency. However it does not specify how synchronisation is implemented. Different synchronisation strategies result in variants of the design pattern and impact quality attributes.

The prototype of the architecture design needs to implement and evaluate the alternative patterns or pattern variants. The rest of extension activities now can be seamlessly integrated with MEMS.

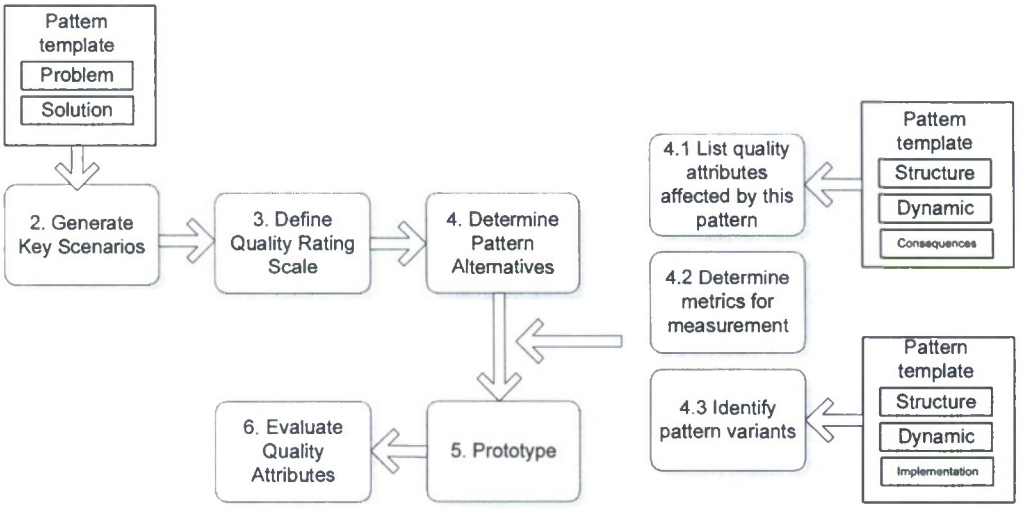

Figure 5-2. MEMS extension for integrating design patterns

5.2 An Example

The performance quality attribute will be used to illustrate how to practice MEMS and its extension for design patterns. The metrics of the performance quality attribute include average response time, throughput and resource utilisation. The stimulus can be the arrival rate under a certain distribution and the response can be the average response time of a group of requests. One generic scenario is to maintain the throughput level when the arrival rate of requests increases. The specified key scenario is to increase concurrency when the number of requests exceeds a threshold.

The active object design pattern aims to enhance concurrency and simplify synchronised access to objects that reside in their own threads of control (Chapter 5 in [Schmidt *et al.* 2000]). The active object design pattern has a description including context, problem, solution, elements in its structure and their dynamic interactions, the implementation generic to any platform, and consequences. Information captured by the design pattern description is not

specific to the key scenario described above, however it can be tailored to the key scenario within the scope of the pattern problem and the pattern context. By further examining the structure and implementation of the pattern, the following quality attributes can be identified.

- Information hiding (modifiability). The elements *Proxy* and *Servant* separate method invocation and execution.
- Intermediary (modifiability). The element *Scheduler* acts as an intermediary that schedules the execution invocation.
- Binding time (modifiability). The active object pattern assumes that requests for the object arrive at the object at runtime. The binding of the client to the proxy, however, is left open in terms of binding time.
- Scheduling policy (performance). The *Scheduler* implements some scheduling policy.
- Multithreading (performance). The *Scheduler* and *Servant* execute in a separate thread from the *Proxy* and the client.
- Synchronisation (performance, scalability). The synchronisation operations performed by the *Scheduler* through the method guard incur performance overhead and the synchronisation strategy can affect scalability when concurrent requests are synchronised.
- Synchronisation (liveliness). The synchronisation operations performed by the *Scheduler* through the method guard have impact on liveliness. Inappropriate implementation of the scheduling policy and the method guard can lead to deadlock, starvation or livelock.

The variants of active object pattern are discussed within the pattern description in [Schmidt *et al.* 2000]. It can be seen from the analysis above that synchronisation impacts performance and liveliness. The concurrency strategy can be implemented at different levels to support synchronisation. Locking oriented mechanisms for concurrency control can be considered pessimistic because the critical resources are locked even though operations may not be in conflict with other executing operations. Variants of the active object pattern include:

- Read only. If the requests are read-only then synchronisation is not necessary, as the states are never updated or changed.
- Read mostly. If most of the requests are read-only and only a small amount of requests are updating states, the implementation can follow the read-mostly design pattern, in which the read and write operation are separated from each other. The state involved in read operations is not loaded until the state is updated by a write operation and then an exception to invalidate the state is received.
- Optimistic Concurrency Control (OCC). In the OCC approach, resources are not locked as it is assumed that they will not be modified by other operations. The execution of an operation with OCC has a validation phase. When an operation T finishes its computation, it enters the validation phase. All operations that conflict with T are restarted by checking the read-sets and the write-sets of transactions. OCC is efficient only if the number of aborted operations is relatively insignificant. Hence, it is cost-effective if the level of conflict is sufficiently low.

These variants of the active object pattern can be prototyped within the scenario implementation. Quality attributes such as modifiability, performance and scalability can be evaluated and measured against the variants identified.

6. Hybrid Mission System Testbed

6.1 Overview

The purpose of the Hybrid MST is to provide infrastructure that enables the investigation and demonstration of distributed computing technologies and concepts in modern airborne mission systems. For example, system latencies, capacities and quality of service are issues that are inherent to different network, system and component configurations.

The Hybrid MST is not a simulation of an airborne mission system; rather it incorporates components similar to those found in such systems. The software consists of the Solaris operating system, the Boeing Australia Software Architecture Framework (SAF), the Solipsys Tactical Display Framework (TDF) and a number of software components developed by AMS to exercise the SAF. While the software normally executes on a Sun Microsystems workstation running Solaris, it can also be compiled and executed on a PC or deployed to any mix of PCs and Sun workstations (except for the Rosetta Adapter, which must run on a PC). Deployment is defined at run-time.

The SAF provides common functionality for the development of systems consisting of distributed components and service oriented architectures. It encapsulates the details of the Common Object Request Broker Architecture (CORBA) middleware and transports. The CORBA specification supplies a set of abstractions and services to address the problems associated with distributed and heterogeneous computing systems. These problems include reliance on programming languages, operating systems, communication protocols and hardware. More detailed information on CORBA can be found in [Henning & Vinoski 1999; CORBA 2006].

The SAF is built on ACE[1], the ADAPTIVE[2] Communication Environment, which mitigates the direct dependence of application software on the underlying operating system. ACE, developed by the Distributed Object Computing (DOC) Group, is an open-source OO framework that implements many core patterns for concurrent communication software. The main application of ACE is the development of high performance, real-time and distributed communication services, with the aim of reducing complexity through higher layer abstractions. An additional abstraction layer, The ACE ORB[3] (TAO[4]), implements the CORBA middleware specification while utilising the patterns and mechanisms of the lower ACE abstraction.

[1] http://www.cs.wustl.edu/~schmidt/ACE.html.
[2] A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment.
[3] Object Request Broker
[4] http://www.cs.wustl.edu/~schmidt/TAO.html.

The SAF, along with ACE, incorporates a platform abstraction layer that normalises environmental function in accordance with standards (e.g. POSIX) and is therefore source and function compatible across multiple platforms, including Solaris and Windows.

The SAF core is based on a CORBA Component Model (CCM) organisational style, but predates CCM. Internal SAF structures are transparent to standard CORBA environments and interfacing mechanisms, allowing for standards-based component integration.

The SAF provides a set of utilities, containers (based on the C++ standard template library (STL)) and patterns that provide general services to components, with the aim of increasing the rate of component development and reducing integration risk. The SAF provides services relevant to distributed computing environments, for example dynamic component deployment (the idea of application containers is supported), event management, concurrency control, streams and system configuration control. The use of design patterns that emphasise distribution and concurrency [Lea 1999] addresses the properties generally required of modern airborne mission systems, for example:

- Performance: latency, throughput, CPU utilisation, memory utilisation and LAN utilisation.
- Availability: robust to failure and efficient recovery and restart.
- Extensibility, modularity, scalability and interoperability.

The custom software developed by DSTO for the Hybrid MST takes the form of software components[5]. This reduces coupling and permits communication via an ORB. The components of the Hybrid MST can be grouped into four main categories (Figure 6-1) and represent: COTS, the stimulation environment, mission computing components and monitoring components. The following overviews components developed by DSTO:

- The stimulation environment provides components that generate traffic to alter the state of the Hybrid MST:
    - A Test Track Generator populates and updates the Track Manager component with random tracks.
    - An Air Vehicle component provides a simple model of an aircraft that maintains the aircraft position.
    - The STAGE[6]/MST Interface enables the Hybrid MST to use the STAGE COTS product for ownship sensor and flight modelling.

[5] Components are '...units of composition with contractually specified interfaces and explicit context dependencies only' [Szyperski 1998, p. 41].

[6] Stimulation Toolkit and Generation Environment

- Representative mission computing components:
  - A Track Manager component maintains the tactical state of the aircraft. It achieves this through a common repository of all tracks in the environment and a collection of capabilities that can be applied to these tracks.
  - An Ownship component maintains the kinematic state of the aircraft.
  - A Rosetta Adapter component provides an interface between the Hybrid MST and Rosetta, a COTS tactical data link gateway.

- Monitoring components provide interfaces for observing the information received and stored by other components of the Hybrid MST:
  - A Track Monitor component periodically accesses and displays the details of all tracks in the Track Manager to an operator.
  - The TDFAdapter is technically not a component, but an active object activated by the Track Monitor. The TDFAdapter provides an interface to the Solipsys TDF application to enable visualisation of tracks contained in the Track Manager.

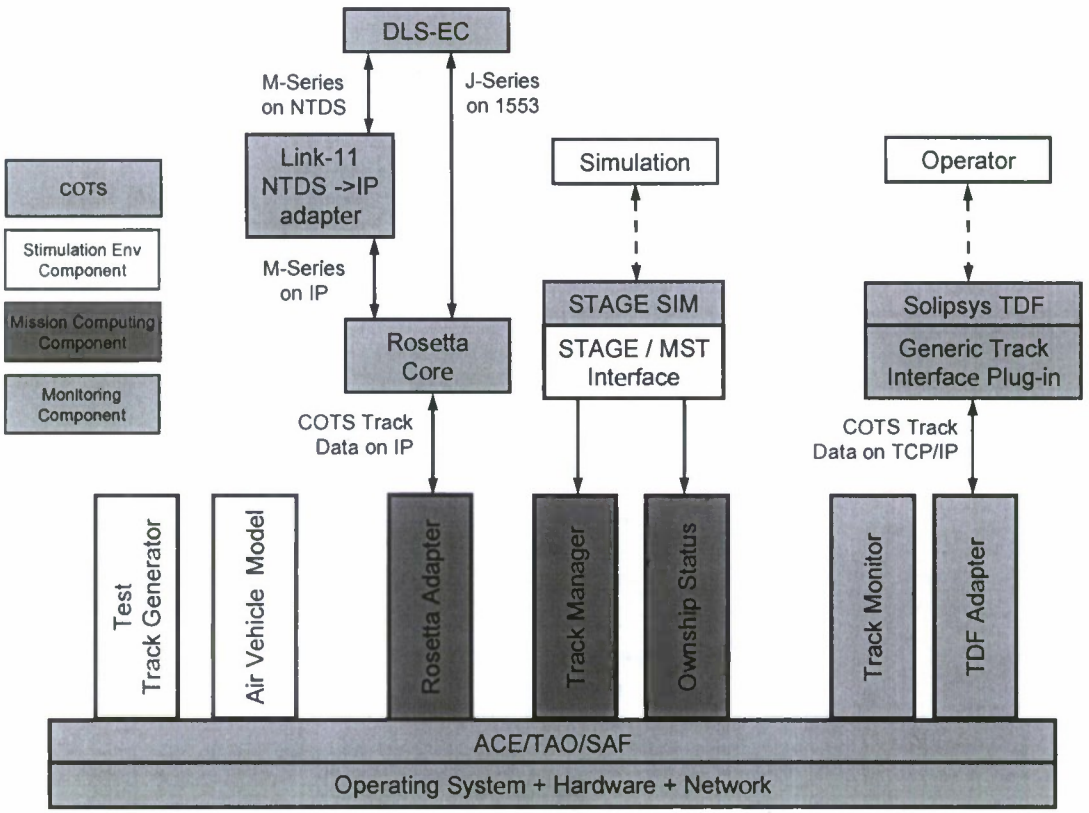

Figure 6-1. Structure of the Hybrid Mission System Testbed

The interaction between the Hybrid MST components is shown in Figure 6-2. Existing components and their interaction with internal and external systems are shown in black, while components and interactions yet to be developed are represented in blue.

The Hybrid MST is a flexible environment for exploring software architectures, and the properties of the SAF and DSTO developed software. This allows integration issues and the performance of various design patterns to be evaluated. The Hybrid MST also provides a research capability to investigate the integration of airborne mission systems into NCW environments, in which it would act as a node in a system of systems network. This section has provided an overview of the Hybrid MST; a more detailed discussion can be found in [Foster *et al.* 2007].

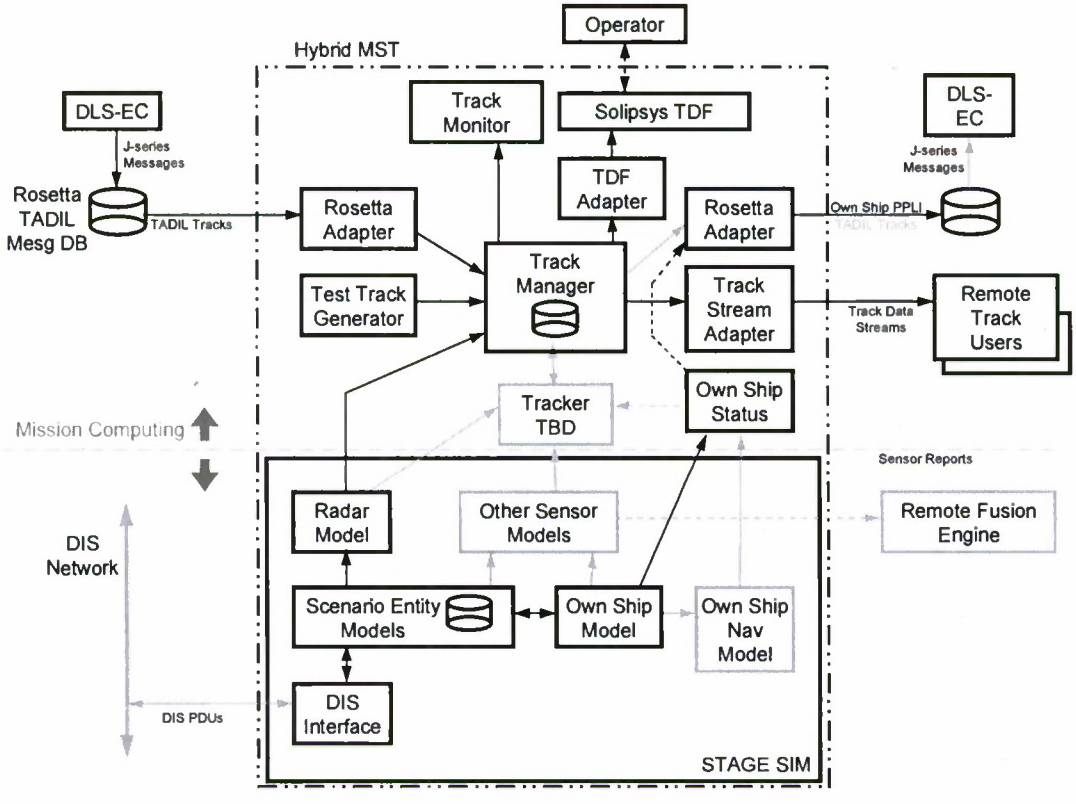

Figure 6-2. Interaction between components in the Hybrid Mission System Testbed (existing components are shown in black and future work is shown in blue)

6.2 Components used for Evaluation

6.2.1 Overview

The major components that will be used for the purpose of evaluating the Hybrid MST are: the Test Track Generator, Track Manager and Track Monitor (including the TDFAdapter). This configuration is shown in Figures 6-3 and 6-4. Together these components form a multithreaded, component-based, distributed application that is able to create, update, manage and use data objects called tracks.

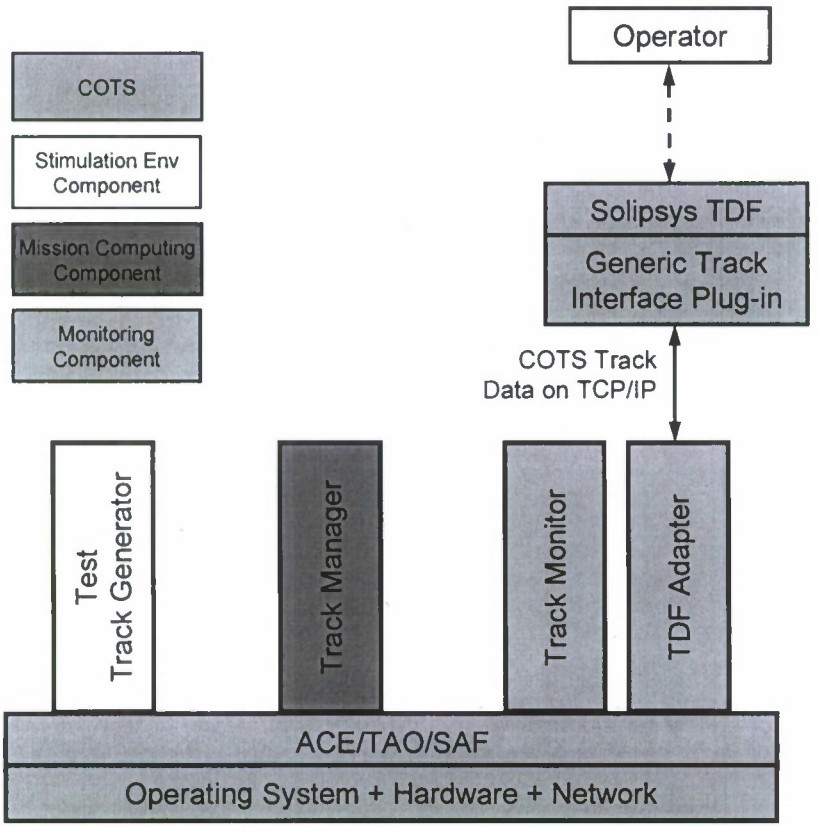

Figure 6-3. The major components used to evaluate the Hybrid Mission System Testbed

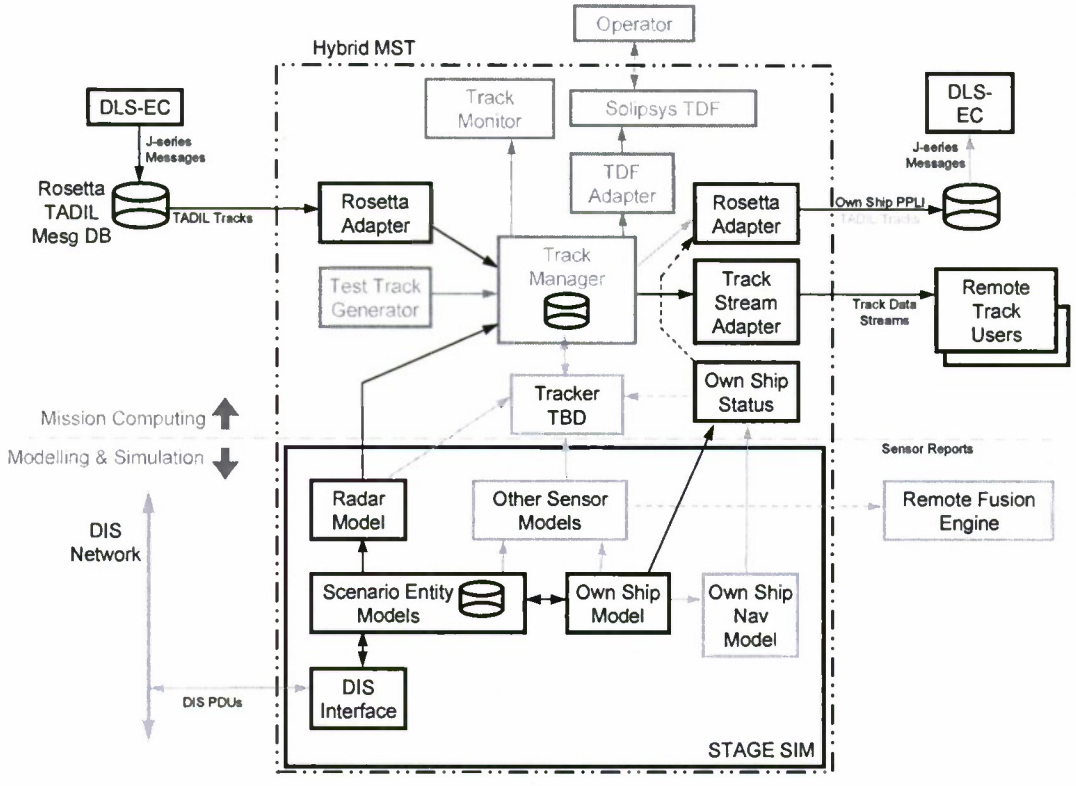

Figure 6–4. Interaction between the components used to evaluate the Hybrid Mission System Testbed (the components to be used for the evaluation are shown in pink)

6.2.2 Test Track Generator

The Test Track Generator component consists of two active objects: a Track Writer and a Track Updater. Using Track Writer and Track Updater active objects enables the two tasks of writing and updating tracks to be performed as concurrently as possible with the one local container in the Test Track Generator.

The Track Writer defines the track details of a track (step 1 in Figure 6–5), which consists of data such as: track identification number, latitude, longitude and status (pending, unknown, assumed-friend, neutral, suspect, hostile and undefined). Tracks are added to the Test Track Generator's local container (step 2 in Figure 6–5), which employs the containment pattern. The Track Writer then invokes the Track Manager's *updateFusedTrackData* method (step 3 in Figure 6–5) to add tracks to the Track Manager (Section 6.2.3).

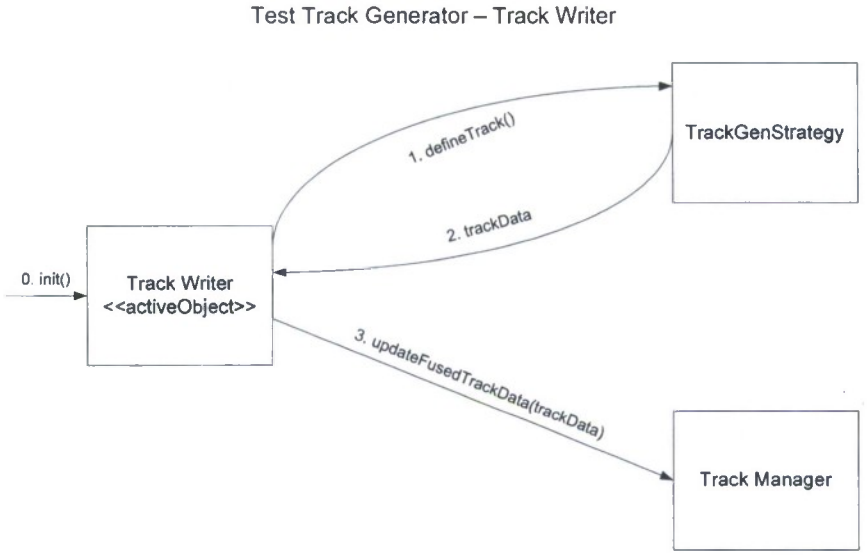

Figure 6–5. Collaboration diagram for writing tracks from the Test Track Generator

The Track Updater (Figure 6–6) is responsible for updating the latitude and longitude of tracks so that if a visual display is used the tracks will move over time. A callback is used to update the Test Track Generator's local manager with new latitude and longitude for each track. Concurrency patterns are used to synchronise reads and writes to the local manager.

When a callback from the Track Manager is invoked by the Track Updater, the *run* method of the Track Updater calls *for_each* (an implementation of iteration patterns) over the local manager to update the Track Manager's container with tracks from the Test Track Generator.

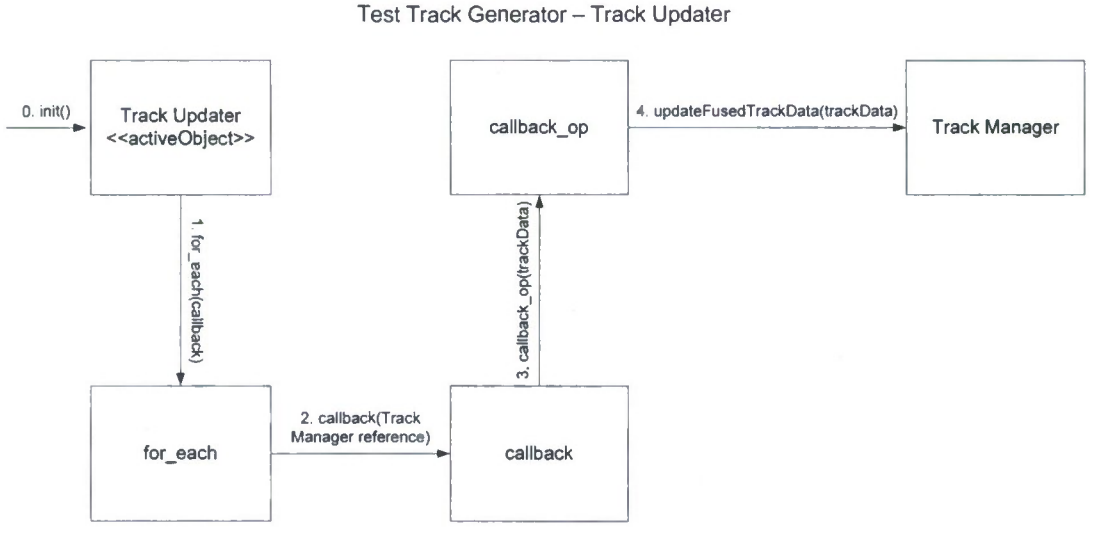

Figure 6–6. Collaboration diagram for updating tracks from the Test Track Generator

6.2.3 Track Manager

The Track Manager maintains a common repository (i.e. containment pattern) of tracks in the environment and provides a collection of capabilities to manage those tracks. The main patterns used in the Track Manager are component home, factory method, active object, leader/follower, evictor, façade, iterator, and scoped locking with read/write semantics implemented through the use of various mutex methods. Other patterns used but not discussed in this report are monitors and channels. Collaboration diagrams are provided (Figures 6–7 and 6–8) that map patterns to aspects of the process of writing and updating tracks and indicate the sequence of events.

The component home pattern is similar to patterns employed by other component technologies that use containers, e.g. *home* in CORBA 3 [OMG 2004], *EJBHome* in Enterprise Java Beans [DeMichiel & Keith 2006] and *container* in .NET [MSDN 2007]. The Track Manager is essentially a component home for tracks and provides a central location to create, store and access tracks in a distributed environment. The Test Track Generator (Section 6.3.2) is able to create or update tracks with a single call to the Track Manager. The Track Monitor (Section 6.3.4) is able to access the tracks through the use of a finder interface which is implemented with callbacks, iterators and locks.

The tracks in the repository are static and the Track Manager relies on updates from the Test Track Generator. The Test Track Generator interacts with the Track Manager through a single event (step 0 in Figures 6–7 and 6–8): a call to the Track Manager's *updateFusedTracks* method (defined in CORBA's Interface Definition Language (IDL)), which performs aspects of the behaviour of the factory method of the component home pattern. The role of the *updateFusedTracks* method is to create a track if it does not exist or update an existing track. This event is handled through a Fused Track Updater, which uses a combination of the active object and leader/follower patterns. The Fused Track Updater separates the track repository update from the general execution of the Track Manger, enabling internal processing and requests from other clients to occur concurrently.

A requirement of the Track Manager is to handle multiple update requests simultaneously. Simply being multithreaded does not guarantee efficiency in the execution or handling of these requests. Therefore, the leader/follower pattern is used as a handoff strategy [Schmidt *et al.* 2000] to enhance system responsiveness to requests. The parallelism of the leader/follower pattern provides for liveliness in performance when updating the Track Manager from multiple track sources. A thread pool is required to accommodate the leader/follower pattern and, therefore, a thread pool (managed by a Pooled Executor) is instantiated within the scope of the Fused Track Updater. The Fused Track Updater is the source of the threads that execute commands generated by input clients (e.g. multiple Test Track Generators). Therefore, there can be many clients generating update events and each event can be handled efficiently with minimal blocking. The purpose of using this pattern is to enable the design of the Track Manager to dynamically adapt to a changing number of input clients.

The Fused Track Updater submits itself to the Pooled Executor for execution. As the Pooled Executor is instantiated within the scope of the Fused Track Updater, the Fused Track Updater is actually submitted to itself for execution (step 2 in Figures 6–7 and 6–8) and the

execute method of the Pooled Executor is called (step 3 in Figures 6–7 and 6–8). The Pooled Executor invokes the *run* method of the Fused Track Updater, which invokes the *uftd* method of the Fused Track Updater (step 4 in Figures 6–7 and 6–8). The remainder of the steps in Figures 6–7 and 6–8 (steps 5 to 21 for writing a new track and steps 5 to 14 for updating an existing track) are all performed in the *uftd* method. Consequently, locks (described below) on the Track Manager's container are placed in the *uftd* method.

To manage the lifecycle of a track object, the Track Manager requires mechanisms for the removal and deactivation of tracks from the repository. Data associated with a track object is volatile and dynamic and becomes obsolete if not updated periodically. Tracks are removed from the Track Manager through a track eviction mechanism (based on an evict time), which employs the active object pattern for parallel execution.

The finder component of the component home pattern is the interface that allows clients to access tracks that are managed by the Track Manager. This interface employs a façade pattern [Gamma *et al.* 1995] that simplifies the use of an iteration process within the context of read/write semantics. The benefit of using the façade pattern is that it makes accessing external representations of tracks independent of the way they are stored within the Track Manager. It also ensures that access to the tracks is managed in a thread-safe manner so that data integrity is guaranteed in multithreaded environments.

The iterator is a design pattern used to access elements of an object without exposing the underlying representation of the object. As such, the Track Manager uses iterators in conjunction with read/write semantics to perform the finder task required by the component home pattern. Internally, the Track Manager uses an STL *map* implementation protected by read/write semantics to store tracks and thus uses STL iterators to traverse the map. Rather than force clients to work with iterators and synchronising mechanisms directly, the Track Manager uses a façade pattern to provide an interface that a client, in conjunction with providing a callback interface, can use to locate and operate on tracks.

While in this case there are no real drawbacks to using the façade pattern to provide the finder interface, there are concurrency issues concerning the liveliness of the Track Manager when using iterators in conjunction with read/write semantics. Each time an iterator is used to iterate over the container, a read lock must be exerted to ensure the container organisation is not altered during reads as this would invalidate the iterator. If there are many clients wishing to access the tracks concurrently then there may be many iterators operating on the container and thus many read locks exerted. If another client wanted to alter the container's organisation by adding or removing a track it would require a write lock, but the write lock would block until all readers have finished. A reduction of liveliness could result if there were many tracks in the container and many clients reading from and writing to the container. If liveliness was to become a performance issue, other methods of providing access to tracks would need to be explored.

The use of callbacks when taking read locks has a latency effect. While this is not an issue for containers with low volatility, if the container in the Track Manager becomes highly volatile an alternate method may need to be considered.

The protection of the Track Manager's container is achieved through extensive use of read/write locking semantics. These semantics are accomplished through the use of scoped locking approaches and patterns.

Read/write locking works by allowing multiple readers to simultaneously access an object without being blocked, whereas a writer must have exclusive access to the object. When a writer requires access to the object all new arriving readers are blocked, with the writer forced to wait until all current readers have released their locks. If there are multiple write requests, access is typically granted in FIFO order with priority given to waiting writers over waiting readers.

Scoped locking through the use of guards (an idiom) is a pattern designed to simplify locking semantics to ensure locks are acquired and released consistently. An implementation of a guarding pattern is utilised, which acquires and releases locks based on scope. This prevents aberrant locking, particularly due to abnormal conditions (e.g. exceptions), or poor programming practices by ensuring that acquired locks are always released. The ACE framework provides *read guard* and *write guard* that implement scope-based semantics through scoped locking patterns.

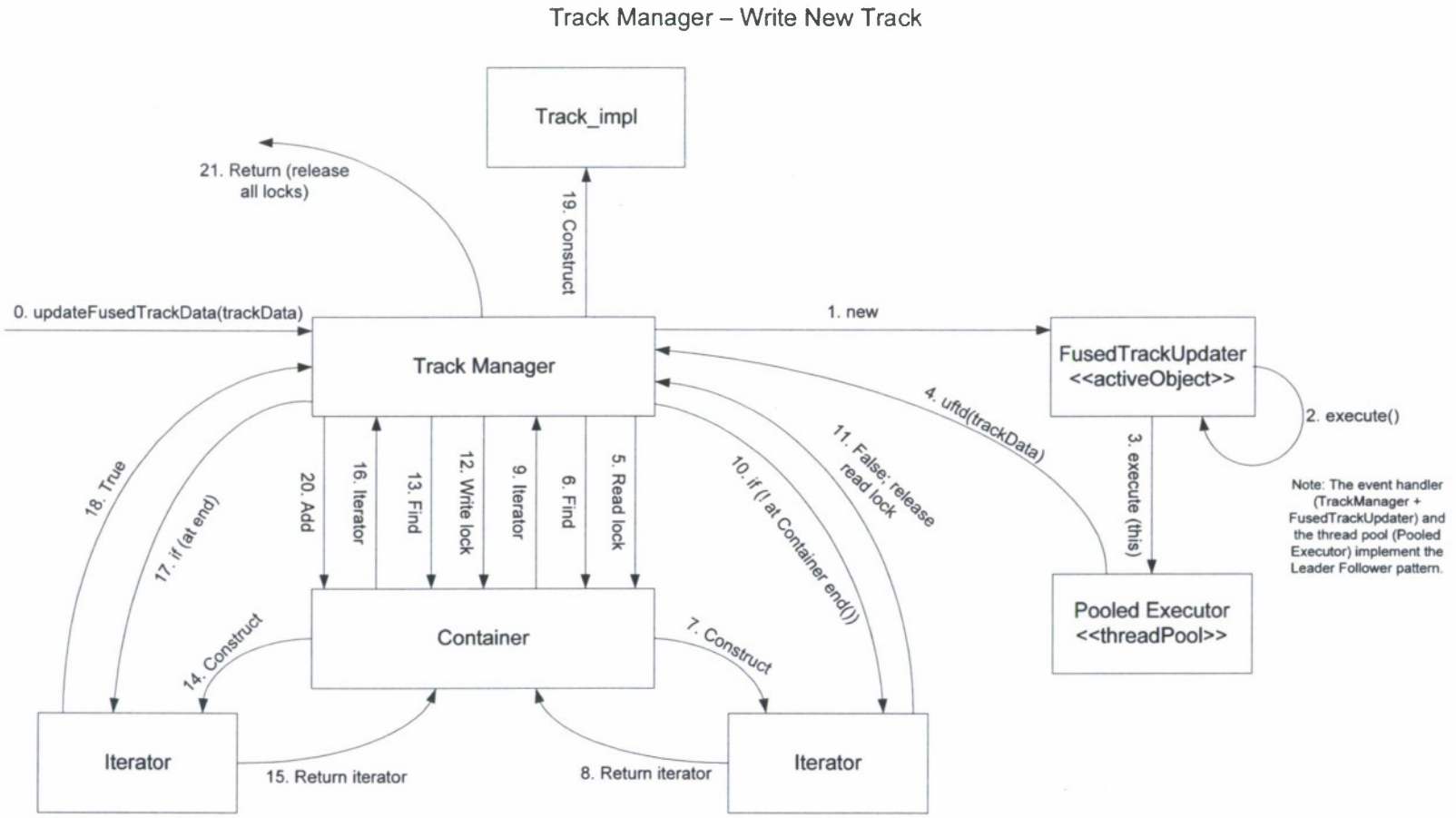

Figure 6–7. Collaboration diagram for writing tracks in the Track Manager

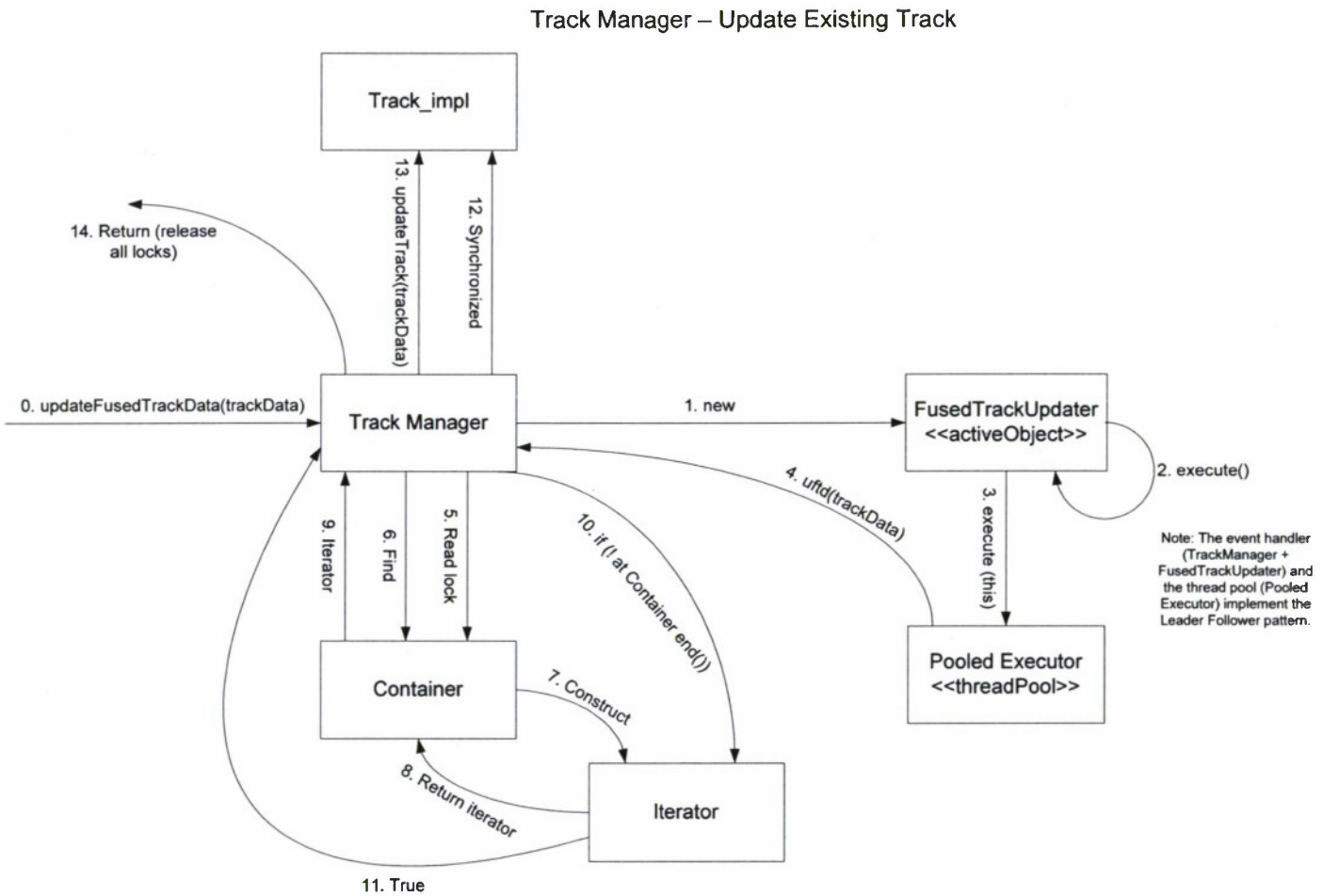

Figure 6–8. Collaboration diagram for updating tracks in the Track Manager

6.2.4 Track Monitor

The Track Monitor is a simple component that periodically accesses and displays the details of all tracks in the Track Manager to an operator (Figure 6–9). The Track Monitor is responsible for activating two active objects (Track Streamer and TDFAdapter), which output information contained in the Track Manager in alternative ways.

The Track Streamer defines an output stream as its means for output, which is a common interface for writing data. This output stream conforms to a push model for writing, indicating that it knows the identity of the receiver before pushing the message. A callback is defined to encapsulate the writing of tracks to this output stream.

The TDFAdapter provides an interface to the Solipsys TDF. This adaptation is encapsulated within a callback to ensure the necessary data conversions take place for accurate representation on the TDF Graphical User Interface (GUI).

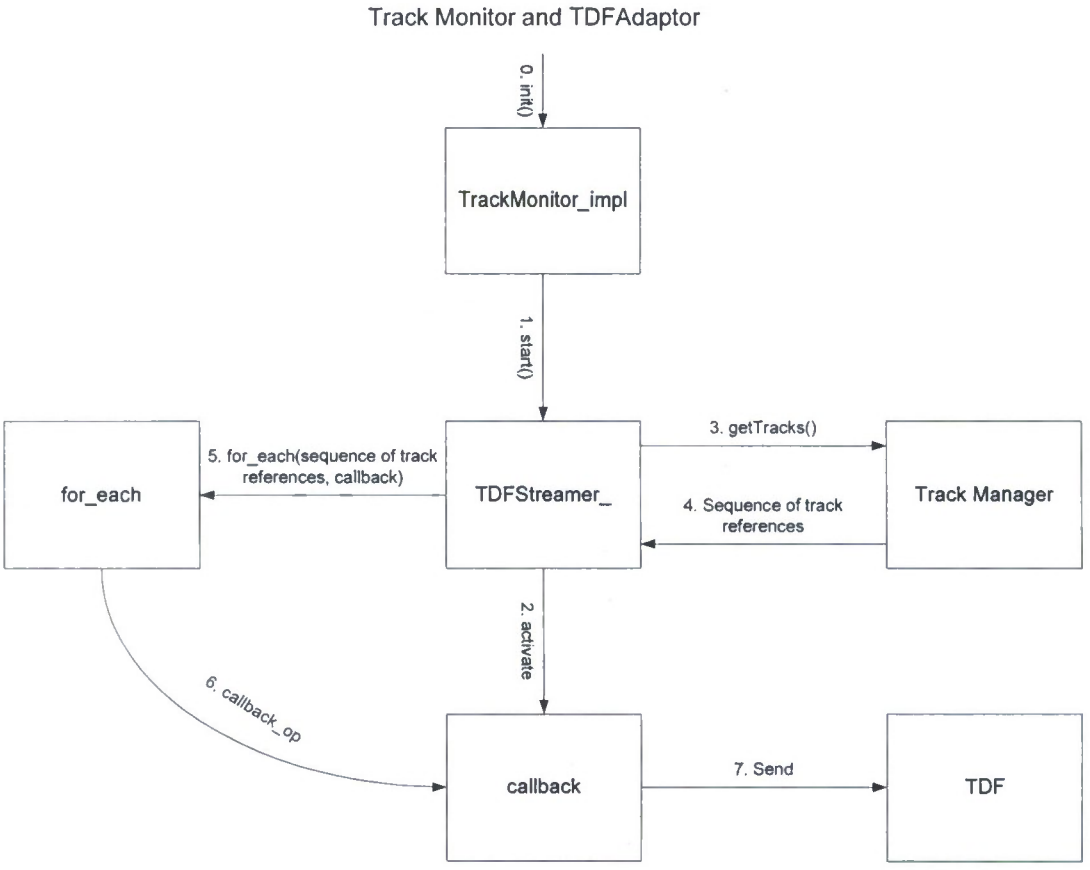

Figure 6–9. Collaboration diagram for displaying tracks

7. Evaluating the Hybrid Mission System Testbed using MEMS

7.1 Mapping MEMS Artefacts, Roles and Tasks

Figure 7-1 maps the architecture design artefacts discussed in Section 4 to elements of the Hybrid MST. It demonstrates that the software components and their interactions with the middleware-based environment determine the key scenarios of the quality attributes. The MEMS approach should be applied within the technology context of these software components.

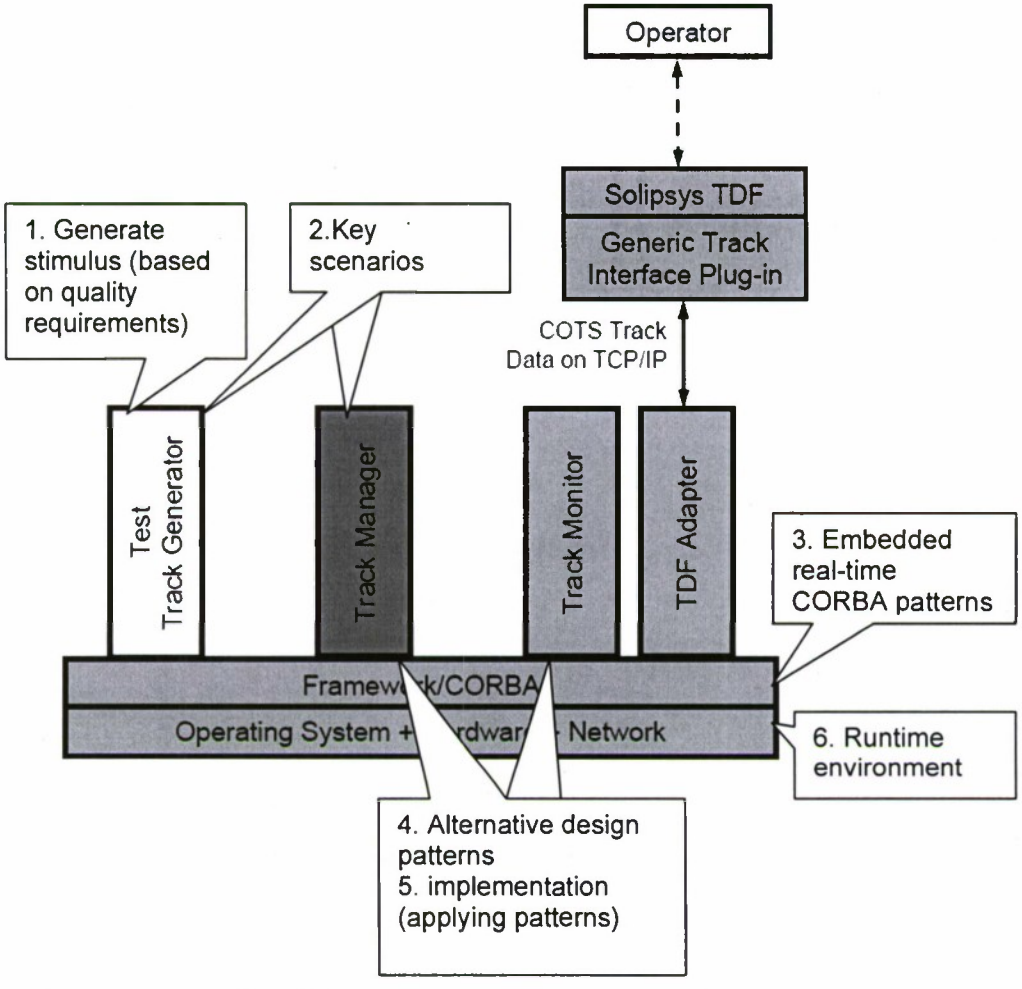

Figure 7-1. Mapping MEMS artefacts to the Hybrid Mission System Testbed

The roles of the generic MEMS approach are further mapped to the context of this architecture evaluation project. The MEMS approach involves activities from two sides, DSTO and NICTA. The leader of this evaluation project at DSTO represents the role of the *stakeholder*. Stakeholders define business goals and provide requirements for the evaluation project. It is worth noting that the requirements are for middleware quality attributes and not for a system deployed and running on the middleware. The role of the architect is fulfilled by both NICTA researchers and the evaluation team at DSTO. NICTA researchers act as the *architect* for MEMS, by extending and customising MEMS to the context of airborne mission systems. NICTA also takes a support role by working with the DSTO evaluation team to plan the evaluation according to the MEMS steps. The evaluation team consists of software architects and researchers from DSTO, who have expertise in airborne mission systems. The evaluation team follows the MEMS steps and steers the evaluation process. The evaluation involves the instrumentation, configuration and further development of the Hybrid MST, running experiments and taking empirical measurements. This task is performed by developers from DSTO. The developers also work closely with the evaluation team by providing feedback and comments on the feasibility of the evaluation plan.

7.2 Identifying Important Architectural Patterns for Evaluation

Section 6.2 discussed the components to be used for the evaluation of the Hybrid MST and a number of design patterns applied in the development of these components. Two key design patterns are applied in the Hybrid MST: active object and leader/follower [Schmidt *et al.* 2000]. These two patterns are used to manage the concurrency of processing track creation and update operations.

The Track Writer and Track Updater (in the Test Track Generator), and the Fused Track Updater (in the Track Manager) are all active objects. An active object processes requests within its own thread of control and is implemented as a runnable object, which is submitted to an executor that has an underlying thread pool. The Track Writer active object generates requests to create a new track and the Track Updater active object updates existing tracks. The operations performed by these two active objects are not orthogonal because they both share the local container in the Test Track Generator and the container in the Track Manager. Creating and updating track requests are two classes of workload that can be generated from the Test Track Generator.

The other source of workload is the request to display tracks from the Track Monitor and TDFAdaptor. The *for_each* method is of concern in terms of the performance evaluation, as it iterates over the set of tracks that is returned by invoking the Track Manager's method *getTracks()*. The requests can be generated by running Track Monitor and TDFAdapter. Multiple instances of these two components can be executed to generate the workload required.

The execution of Fused Track Updater is multithreaded, supported by a Pooled Executor in which the leader/follower design pattern is applied.

These three sources of concurrent requests incur contention over the container inside the Track Manager. This container has a STL data structure to store the tracks in memory. Concurrent access to the container is controlled by:

- A synchronisation macro implemented using a locking mechanism called a read/write mutex (i.e. read locks and write locks can be taken separately).
- A mutually exclusive lock with acquire and release semantics (i.e. there is no concept of *read* or *write* and all other threads are prevented from accessing a track even if just reading).

In the SAF, a lock cannot be upgraded or downgraded. Regardless of whether a read or write lock is held, a lock is obtained before its execution can be performed in the container. Figure 7-2 shows the locks and concurrent requests from three sources. This indicates that the performance and scalability in terms of the throughput depend on two factors: the workload from these three sources and the locking strategy inside the Track Manager.

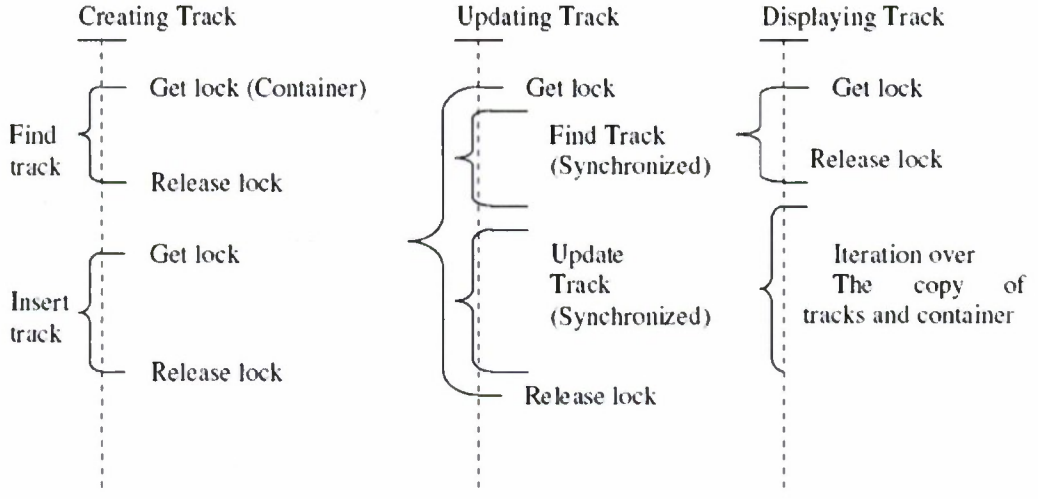

Figure 7-2. Locks and concurrent load

For exclusive locking, the time spent on blocking is critical to the overall performance, because a lock needs to be acquired for each operation of interest and it involves the iteration over each track in the container. An operation can be blocked for a calling thread up to *max_wait* time, when the thread requests a lock.

Based on above understanding of the Hybrid MST, the optimisation of quality attributes within this architecture is likely to be focused on locking management and optimisation for exclusive locks. Therefore, this forms an important scenario for quality attributes that include performance, scalability, modifiability and liveliness.

The limitation of the SAF on optimising the locking mechanism needs to be examined. In particular, the components of the SAF that might be affected or required to evolve need to be identified. This could be an important output of evaluating the Hybrid MST by identifying the

limitations of the current middleware infrastructure and evaluating the cost of improving this through the modifiability quality attribute.

7.3 Outline of the Evaluation Plan

The following is an outline of an evaluation plan for the Hybrid MST:

- Determine the quality attributes of interest. An initial list of quality attributes has been identified:
- Performance in terms of response time of request execution and overall throughput.
- Scalability in terms of how the system performs when the number of tracks from one source is increased or the number of track sources is increased.
- Modifiability in terms of the development cost when additional functionality is added.
- Liveliness in term of the occurrence of deadlocks, starvation and livelock.
- Construct the key scenarios for each quality attribute. These key scenarios directly affect the design and implementation of the evaluation testbed.
- Define the metrics for measurements of each quality attribute. These metrics can be identified based on analytical models, which provide guidance regarding the impact of changing analytical model parameters.
- Define the test cases and the measurements for each test case. This is integrated with the prototype step within MEMS.
- Collect and analyse results. Note that this step may lead to refinement of the test case if the results are not as expected or the results are not complete. More results will then be collected and analysed.
- Document the results, possibly using an architecture knowledge management tool developed by NICTA [Ali-Barbar *et al.* 2005].

A complete evaluation plan will be developed based on the above outline.

8. Summary

The ADF is acquiring airborne mission systems that incorporate component-based and distributed computing systems. These systems are built on middleware, which is a broad class of software infrastructure technologies that use high-level abstractions to simplify the construction of distributed systems. Therefore, middleware significantly impacts the overall quality of the system.

As a technical evaluator of ADF acquisitions, DSTO has developed a research program to investigate methods and techniques to evaluate component-based and distributed software architectures. This research is being conducted under AMS LRR Task 06/075 and involves collaboration between DSTO and NICTA.

Further research will involve: the development of a detailed evaluation plan; instrumentation and configuration of the Hybrid MST to enable the evaluation plan to be implemented; conducting experiments for each of the scenarios in the evaluation plan; and analysing the results obtained from these experiments. The evaluation results and the patterns used in the Hybrid MST may be documented using an architecture knowledge management tool also developed at NICTA.

9. References

[Ali-Babar & Gorton 2004] Ali-Babar, M. & Gorton, I. (2004) Comparison of Scenario-Based Software Architecture Evaluation Methods, in the *Proceedings of the 11th Asia-Pacific Software Engineering Conference (ASPEC'04)*, pp. 600–7.

[Ali-Babar *et al.* 2005] Ali-Babar, M., Wang, X. & Gorton, I. (2005) PAKME: A Tool for Capturing and Using Architecture Design Knowledge, in the *Proceedings of the 9th International Multitopic Conference*, pp. 1–6, IEEE.

[Allen *et al.* 2002] Allen, R., Vestal, S., Cornhill, D. & Lewis, B. (2002) Using an Architecture Description Language for Quantitative Analysis of Real-Time Systems, in the *Proceedings of the 3rd International Workshop on Software and Performance*, Rome, Italy, pp. 203–10.

[Bachmann *et al.* 2003] Bachmann, F., Bass, L. & Klein, M. (2003) *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design*, CMU/SEI-2003-TR-004, Carnegie Mellon University Software Engineering Institute, United States.

[Barbacci *et al.* 1995] Barbacci, M., Klein, M., Longstaff, T. A. & Weinstock, C. B. (1995) *Quality Attributes*, CMU/SEI-95-TR-021, Carnegie Mellon University Software Engineering Institute, United States.

[Bass *et al.* 2003] Bass, L., Clements, P. & Kazman, R. (2003) *Software Architecture in Practice*, 2nd ed., Addison-Wesley Professional.

[Bass *et al.* 2001] Bass, L., John, B. E. & Kates, J. (2001) *Achieving Usability Through Software Architecture*, CMU/SEI-2001-TR-005, Carnegie Mellon University Software Engineering Institute, United States.

[Bengtsson *et al.* 2004] Bengtsson, P., Lassing, N., Bosch, J. & Vliet, H. (2004) Architecture-Level Modifiability Analysis (ALMA), *Journal of Systems and Software*, **69**(1–2), pp. 129–47.

[Boehm & In 1996] Boehm, B. & In., H. (1996) Identifying Quality-Requirement Conflicts, *IEEE Software*, **13**(2), pp. 25–35, IEEE Computer Society.

[CORBA 2006] *Overview of CORBA* (CORBA) (2006) (accessed 13 July 2007), http://www.cs.wustl.edu/~schmidt/corba-overview.html.

[Clements *et al.* 2001] Clements, P., Kazman, R. & Klein, M. (2001) *Evaluating Software Architectures*, Addison-Wesley Professional.

[DMO 2004] Defence Materiel Organisation (DMO) (2004) *Defence Electronic Systems Sector Strategic Plan*, Defence Publishing Service, Department of Defence, Canberra, Australia.

[DeMichiel & Keith 2006] DeMichiel, L. & Keith, M. (2006) *Enterprise JavaBeans Version 3.0 (Specification)*, Sun Microsystems.

[DFW 2004] Directorate of Future Warfighting (DFW) (2004) *Enabling Future Warfighting: Network Centric Warfare*, ADDP–D.3.1, Defence Publishing Service, Department of Defence, Canberra, Australia.

[Dobrica & Niemela 2002] Dobrica, L. & Niemela, E. (2002) A Survey on Software Architecture Analysis Methods, *IEEE Transactions on Software Engineering*, **28**(7), pp. 638–53, IEEE.

[Foster *et al.* 2007] Foster, K., Iannos, A., Lawrie, G., Temple, P. & Tobin, B. (2007) *Exploring Net Centric Architectures using the Net Warrior AEW&C Node*, DSTO-TR-XXXX (draft), Defence Science & Technology Organisation, Edinburgh, Australia.

[Gamma *et al.* 1995] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

[Gorton *et al.* 2003] Gorton, I., Liu, A. & Brebner, P. (2003) Rigorous Evaluation of COTS Middleware Technology, *IEEE Computer*, **36**(3), pp. 50–5, IEEE Computer Society.

[Henning & Vinoski 1999] Henning, M. & Vinoski, S. (1999) *Advanced CORBA Programming with C++*, Addison-Wesley Professional.

[Kanoun *et al.* 1997] Kanoun, K., Kaaniche, M. & Laprie, J.-P. (1997) Qualitative and Quantitative Reliability Assessment, *IEEE Software*, **14**(2), p 77–87, IEEE Computer Society.

[Kazman *et al.* 1994] Kazman, R., Bass, L., Webb, M. & Abowd, G. (1994) SAAM: A Method for Analyzing the Properties of Software Architectures, in the *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, pp. 81–90.

[Kazman *et al.* 2000] Kazman, R., Carrière, S. J. & Woods, S. G. (2000) Toward a Discipline of Scenario-Based Architectural Engineering, *Annals of Software Engineering*, **9**(1–4), pp. 5–33, Springer, Netherlands.

[Kazman *et al.* 1998] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. & Carriere, J. (1998) The Architecture Tradeoff Analysis Method, in the *Proceedings of the Fourth IEEE International Conference on Engineering Complex Computer Systems (ICECCS'98)*, 10 to 14 August, pp. 68–78, IEEE.

[Kontio 1996] Kontio, J. (1996) A Case Study in Applying a Systematic Method for COTS Selection, in the *Proceedings of the 18th International Conference on Software Engineering*, 25 to 29 March, Berlin, Germany, pp. 201–9, IEEE Computer Society.

[Lassing 1999] Lassing, N., Rijsenbrij, D. & Vliet, H. (1999) On Software Architecture Analysis of Flexibility, Complexity of Changes: Size Isn't Everything, in the *Proceedings of the 2nd Nordic Software Architecture Workshop (NOSA'99)*.

[Lawlor & Vu 2003] Lawlor, B. & Vu, L. (2003) *A Survey of Techniques for Security Architecture Analysis*, DSTO-TR-1438, Defence Science & Technology Organisation, Edinburgh, Australia.

[Lea 1999] Lea, D. (1999) Concurrent Programming Java: Design Principles and Patterns, 2nd ed., Addison-Wesley Professional.

[Liu & Gorton 2003] Liu, A. & Gorton, I. (2003) Accelerating COTS Middleware Acquisition: The i-Mate Process, *IEEE Software*, **20**(2), pp. 72–9, IEEE Computer Society.

[MSDN 2007] Microsoft Developer Network (MSDN) (2007) *.NET Framework 3.0* (accessed 16 July 2007), http://msdn2.microsoft.com/en-us/netframework/default.aspx.

[Nandagopal 2006] Nandagopal, N. (2006) Systems Integration Challenges for Defence, *Defence Magazine*, October, pp. 26–7, Department of Defence, Canberra, Australia.

[Nord & Tomayko 2006] Nord, R. L & Tomayko, J. E. (2006) Software Architecture-Centric Methods and Agile Development, *IEEE Software*, **23**(2), pp. 47-53, IEEE Computer Society.

[OMG 2004] Object Management Group (OMG) (2004) Common Object Request Broker Architecture: Core Specification Version 3, Object Management Group.

[Paravisini 2003] Paravisini, B. (2003) The Complexity of Weapon Systems, *Doctrine*, **1**, pp. 40–2.

[Schmidt *et al.* 2000] Schmidt, D., Stal, M., Rohnert, H. & Buschmann, F. (2000) Pattern-Orientated Software Architecture: Patterns for Concurrent and Networked Objects, vol. 2, Wiley.

[Smith *et al.* 2004] Smith, J., Egglestone, G., Farr, P., Moon, T., Saunders, D., Shoubridge, P., Thalassoudis, P. & Wallace, T. (2004) *Technical Risk Assessment of Australian Defence Projects*, DSTO-TR-1656, Defence Systems Analysis Division, Defence Science & Technology Organisation, Canberra, Australia.

[Szyperski 1998] Szyperski, C. (1998) Component software: Beyond Object-Oriented Programming, Addison-Wesley.

[Tang *et al.* 2004] Tang, D., Kumar, D., Duvur, S. & Torbjornsen, O. (2004) Availability Measurement and Modeling for an Application Server, in the *Proceedings of the International Conference on Dependable Systems and Networks*, 28 June to 1 July, pp. 669–78.

[Voelter *et al.* 2004] Voelter, M., Kircher, M. & Zdun, U. (2004) Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware, Wiley.

[Wait 2006] Wait, P. (2006) *Weapons Projects Misfire on Software*, Government Computer News, 3 July.

[Williams & Smith 2002] Williams, L. G. & Smith, C. U. (2002) PASA: A Method for the Performance Assessment of Software Architectures, in the *Proceedings of the 3rd International Workshop on Software and Performance*, 24 to 26 July, Rome, Italy, pp. 179–89, ACM Press.

[Wilcock *et al.* 2001] Wilcock, G., Totten, T., Gleave, A. & Wilson, R. (2001) The Application of COTS Technology in Future Modular Avionic Systems, *Electronics & Communication Engineering Journal*, pp. 183–92.

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

Developing an Evaluation Method for Middleware-Based Software Architectures of Airborne Mission Systems

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Kate Foster, Jenny Liu and Adam Iannos

5. CORPORATE AUTHOR

DSTO Defence Science and Technology Organisation
506 Lorimer St
Fishermans Bend Victoria 3207 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TR-2204	AR-014-310	Technical Report	July 2007

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
2008/1041870	07/245	Long Range Research	36	41

13. URL on the World Wide Web

http://www.dsto.defence.gov.au/corporate/reports/DSTO-TR-2204.pdf

14. RELEASE AUTHORITY

Chief, Air Operations Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS Yes

18. DSTO RESEARCH LIBRARY THESAURUS http://web-vic.dsto.defence.gov.au/workareas/library/resources/dsto_thesaurus.shtml

middleware; airborne mission systems; software architecture; software evaluation

19. ABSTRACT

The Australian Defence Force (ADF) is acquiring airborne mission systems that incorporate component-based and distributed computing systems. Such systems are built on middleware technologies. As DSTO is responsible for technically evaluating ADF acquisitions, one area of research in the Air Operations Division is the evaluation of middleware-based software architectures. In order to conduct this research, DSTO and NICTA have collaborated to extend NICTA's middleware evaluation method and apply it to the airborne mission systems domain.